Randomized Renaming in Shared Memory Systems

Petra Berenbrink*, André Brinkmann†, Robert Elsässer‡, Tom Friedetzky§, Lars Nagel†

* *School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, Canada.*
† *Zentrum für Datenverarbeitung, Johannes Gutenberg-Universität Mainz, 55099 Mainz, Germany.*
‡ *Department of Computer Sciences, University of Salzburg, 5020 Salzburg, Austria.*
§ *School of Engineering and Computing Sciences, Durham University, Durham, DH1 3LE, U.K.*

***Abstract*—Renaming is a task in distributed computing where n processes are assigned new names from a name space of size m. The problem is called tight if $m = n$, and loose if $m > n$. In recent years renaming came to the fore again and new algorithms were developed. For tight renaming in asynchronous shared memory systems, Alistarh *et al.* describe a construction based on the AKS network that assigns all names within $\mathcal{O}(\log n)$ steps per process. They also show that, depending on the size of the name space, loose renaming can be done considerably faster. For $m = (1+\epsilon)\cdot n$ and constant ϵ, they achieve a step complexity of $\mathcal{O}(\log\log n)$.**

In this paper we consider tight as well as loose renaming and introduce randomized algorithms that achieve their tasks with high probability. The model assumed is the asynchronous shared memory model against an adaptive adversary. Our algorithm for loose renaming maps n processes to a name space of size $m = (1+2/(\log n)^{\ell})\cdot n = (1+o(1))\cdot n$ performing $\mathcal{O}(\ell\cdot(\log\log n)^2)$ test-and-set operations. In the case of tight renaming, we present a protocol that assigns n processes to n names with step complexity $\mathcal{O}(\log n)$, but without the overhead and impracticality of the AKS network. This algorithm utilizes modern hardware features in form of a counting device which is also described in the paper. This device may have the potential to speed up other distributed algorithms as well.

I. Introduction

Renaming is a task in distributed computing in which processes are assigned distinct names from a new and usually small name space. The number of processes is denoted by n, the size of the name space by m. The problem is called tight if $m = n$ and loose if $m > n$. Dependent on the model, the processes (synchronously) communicate via messages or have asynchronous access to shared memory. In the former case, one is interested to restrict the number of communication rounds and possibly the size of the messages, in the latter case, in the step complexity which is the maximum number of accesses to the shared memory by any process.

In recent years renaming gained new popularity and several papers appeared that investigated renaming in the message-passing model ([1], [2], [3]) and in the shared memory model ([4], [5], [6], [7], [9]). Assuming the asynchronous shared memory model, the authors of [7] describe a construction based on the AKS network that assigns all names to a tight name space within $\mathcal{O}(\log n)$ steps per process. For loose renaming in the same model, it is shown in [9] that $\mathcal{O}(\log\log n)$ steps are sufficient to provide n processes with distinct names from a name space of size $(1+\epsilon)\cdot n$ where ϵ is a constant.

In this paper we consider tight as well as loose renaming in the shared memory model. The presented algorithms use random bits and achieve their tasks with high probability[1].

For tight renaming, our algorithm has a step complexity of $\mathcal{O}(\log n)$, asymptotically equal to the algorithm of Alistarh *et al.* [7] while avoiding the overhead of an AKS network. In order to achive this result, the names must be stored in a special type of hardware register with an integrated counting device.

Our two algorithms for loose renaming map n processes to a name space of size $m = (1+\epsilon)\cdot n$ *w.h.p.* where $\epsilon = o(1)$. The first algorithm requires a name space of size $m = (1+2/(\log\log n)^{\ell})\cdot n$ and has a step complexity of $\mathcal{O}((\log\log n)^{\ell})$. For the second algorithm, the size of the name space is only $m = (1+2/(\log n)^{\ell})\cdot n$ and the step complexity $\mathcal{O}((\log\log n)^2)$. To the best of our knowledge, these are the first algorithms that achieve almost tight renaming (i. e., with only a sublinear addition of names) in poly-double-logarithmic time.

The remainder of the paper is structured as follows: After a discussion of the related work, the model and a set of tools are described in Section II. The tools include the special hardware register which is used by the algorithm for tight renaming. This algorithm is stated and analysed in Section III, the algorithm for loose renaming in Section IV. The paper is summarized and concluded in Section V.

A. Related Work

There is a substantial amount of work on algorithms for renaming in different models. For the purpose of this

[1] An event $\mathcal{A}$ occurs *with high probability* (*w.h.p.*) if $\mathbf{Pr}[\mathcal{A}] \geq 1 - 1/n^c$ for some constant $c > 0$.

section we only consider results in the shared memory model using test-and-set registers, similar to our model. Additionally, we focus on randomised algorithms; for an overview of deterministic algorithms we refer the reader to [10]. We distinguish between *loose* and *tight* renaming. In loose renaming the name space is larger than the number of processes, whereas the size of the name space equals the number of processes in the case of tight renaming. In the case of *adaptive* renaming the number of processes is not known in advance.

Loose Renaming: The authors of [11] were the first to use randomization for loose renaming. They assume that test-and-set registers are implemented with read-write registers. They present an algorithm that assumes a name space of size $(1+\epsilon)\cdot n$ for a constant positive ϵ. The expected runtime of their algorithm is $\mathcal{O}(M\log^2 n)$, where M is the size of the initial name space. In [4] the authors propose an adaptive implementation of test-and-set registers with read-write registers. Based on that implementation, they present a randomized loose renaming algorithm which, *w.h.p.*, requires $\mathcal{O}(k\log^4 k/\log^2(1+\epsilon))$ steps using a name space of size $(1+\epsilon)\cdot k$. This result was further improved in [12] where the authors present operations for implementing test-and-set with a step complexity of $\mathcal{O}(\log^* k)$ for contention k. The authors of [13] obtained strong long-lived randomized renaming with amortized step complexity $\mathcal{O}(n\log n)$. The step complexity is defined as the maximum number of steps that any process performs in order to find a name.

In [9] the authors assume that the test-and-set registers are given in hardware. They consider loose renaming where the name space is linear in the number of processes. First they assume that n, the number of processes, is known in advance, and present a renaming algorithm with $\mathcal{O}(\log\log n)$ step complexity. Then they present an adaptive algorithm with step complexity $\mathcal{O}((\log\log k)^2)$, where k is the number of processes competing for a name. Both bounds hold with high probability against a strong adaptive adversary. Finally, they show an $\Omega(\log\log n)$ expected time lower bound on the complexity of randomized renaming using test-and-set operations and linear space. Implementing their test-and-set operation would increase the step complexity by a multiplicative $\mathcal{O}(\log\log k)$ and the error terms in their high probability guarantees would become inversely logarithmic rather than inversely polynomial.

For deterministic algorithms, in comparison, the lower bound is known to be $\Omega(n)$ and, thus, exponentially worse [9].

Tight renaming: The authors of [4] present a tight renaming algorithm with a total step complexity of $\mathcal{O}(n\log^3 n)$. In [7] the authors give two new randomized renaming algorithms which work in the presence of an adaptive adversary. The first algorithm has a step complexity of $\mathcal{O}(\log^2 n)$ if the test-and-set registers are implemented in hardware. The second algorithm transforms any sorting network into an adaptive renaming protocol with an expected step complexity cost equal to the depth of the sorting network. Using an AKS sorting network, this gives a strong adaptive renaming algorithm with step complexity $\mathcal{O}(\log k)$. This approach has the disadvantage that the depth of the AKS network is logarithmic but with a rather unwieldy constant, not to mention the complicated structure of an AKS. The approach also needs a large amount of test-and-set registers since the width of the network equals the initial name space of the processes. The authors show that the later result is asymptotically optimal.

Deterministic algorithms for tight renaming, on the other hand, have a step complexity of $\Theta(n)$ [9].

II. Preliminaries

A. Model

The considered machine model is the asynchronous shared memory model with concurrent reads and concurrent writes (CRCW). The processes follow an algorithm composed of steps. Any number of processes may fail by crashing, and a failed process does not perform further steps in the execution. The order in which processes perform steps and their crashes are controlled by an adversary. We assume an adaptive adversary that is allowed to see the state of all processes (including the results of coin flips) when making its scheduling choices.

The asynchronous shared memory contains the name space with m names and can be accessed by all n processes. Besides the name space, additional memory can be used as temporary memory. Like in [9], each name is stored in a test-and-set (`TAS`) register that can be concurrently tested by several processes, but only *won* by one process.

For our tight renaming algorithm, we use a special hardware register, called τ-register. It includes a counting device with `TAS` bits, i.e. `TAS` registers consisting of only one bit. In each step each process is allowed to test at most one `TAS` register or `TAS` bit. If the process wins a `TAS` register or bit, it will get the name in it. Like in other papers, *e.g.* [9], we assume that concurrent accesses to the same `TAS` register or `TAS` bit can be executed in one step and that every name and address can be read or written in one step. Some of these names and numbers have $\log(n)$ bits (or more). Likewise we assume that a processor's registers and instructions can handle numbers of this size and run processor instructions like `xor` and `popcnt` on these numbers in $\mathcal{O}(1)$.

The τ-register and the counting device are described in more detail in the following two sections.

B. τ-register

In order to efficiently calculate tight renaming, our algorithm depends on special hardware registers, called *τ-registers*. Each of these registers has two parts: (i) a set of τ `TAS` registers that contain the names, (ii) a counting device managing $2\log(n)$ `TAS` bits. The counting device counts the number of `TAS` bits set and allows at most τ of them to be set. Any process that wants to get a name has to win one of the $2\log(n)$ `TAS` bits first. After winning a `TAS` bit, the process systematically goes through the `TAS` registers, until it wins one of them, and retrieves the name. It must win one of the `TAS` register because there are exactly τ of them and at most τ processes that are allowed to search.

As the `TAS` registers and the search are straightforward, we only have a closer look at the counting device.

C. Counting Device

The counting device is composed of $2\log n$ individual `TAS` bits and can restrict the number of 1-bits to any positive threshold $\tau \leq 2\log n$. We assume that all individual bits of a τ-register have the same clock as input and that it is possible to read all $2\log n$ individual bits within one operation. The register operates in clock cycles that are divided in phases. The synchronisation of the bits permits that supernumerary `TAS` bits can be unset before the counting device is accessed again by new processes.

However, we do not make any assumptions about the arrival or the order of the requests. Processes can use different clocks and send their requests asynchronously at any time. Yet, since requests are only answered in a certain phase, the processing may start with a (constant) delay. The implementation of a τ-register is based on the following algorithm which represents one clock cycle:

```
1:  allowed_bits ← τ − popcnt(in_reg)
2:  for i ∈ {1, ..., 2log(n)} in parallel do
3:      processes test-and-set bit b_i
4:  if τ < popcnt(in_reg) then
5:      util_reg_0 ← out_reg xor in_reg
6:      for i ∈ {1, ..., 2log(n)} in parallel do
7:          util_reg_i ← util_reg_0 << (i − 1)
8:          if popcnt(util_reg_i) = allowed_bits then
9:              if bt(util_reg_i, 1) then
10:                 util_reg_i ← util_reg_i >> (i − 1)
11:                 out_reg ← out_reg or util_reg_i
12:     in_reg ← out_reg
13: else
14:     out_reg ← in_reg
```

The counting device has two main registers: `in_reg` and `out_reg`. The register `in_reg` contains the `TAS` bits the processes access. The bits of register `out_reg` can be read by the processes to check whether they have really won their respective `TAS` bit. After each execution, both registers are updated such that exactly those bits are set in `in_reg` that have been won by processes and that `out_reg` is an exact copy of `in_reg`. Aside from these two registers, $2\log n + 1$ auxiliary registers $\texttt{util\_reg}_i$, $i = 0, ..., 2\log n$, are needed.

A clock cycle is divided in two phases, the first one covers lines 1–3, the second lines 4–14: In the first line, the algorithm determines the number of bits the register `in_reg` is short of the threshold τ. Then, in lines 2–3, the `TAS` bits of `in_reg` parallelly handle requests of processes. Every request to a `TAS` bit b_i fails if b_i is already set to 1. If bit b_i is unset and if there is at least one request to b_i, b_i will be (preliminarily) set by exactly one of the processes. All other requests to b_i also fail.

If the threshold τ is exceeded in line 4, $(\texttt{popcnt}(\texttt{in\_reg}) - \tau)$ many of the new bits have to be removed. For this purpose, $\texttt{util\_reg}_0$ is prepared in line 5 as a copy of `in_reg` without the old bits, i.e. the bits set prior to this cycle. The algorithm then shifts $\texttt{util\_reg}_0$ by every possible number of bits (line 7) and selects the only resulting bit array $\texttt{util\_reg}_i$ which has both, the correct number of new bits (line 8) and a 1-bit in the first position (line 9). (The first bit is tested using the instruction $\texttt{bt}(\texttt{util\_reg}_i, 1)$.) $\texttt{util\_reg}_i$ is shifted back (line 10) and combined with the old bits in `out_reg` (line 11). The resulting bit array having exactly τ bits, $\tau - \texttt{allowed\_bits}$ old and `allowed_bits` new bits, is stored in `out_reg` (line 11) and `in_reg` (line 12).

If the threshold τ is not exceeded in line 4, `in_reg` can simply be copied to `out_reg` (line 14).

A process that won a `TAS` bit (in line 3) has to check whether this `TAS` bit was later unset (in line 12). It can be certain that the `TAS` bit is unset as soon as it is unset in `in_reg`, and it can be certain to have won it, once the according bit has also been set in `out_reg`. In the latter case, it can immediately start searching the `TAS` registers for a free name.

Each step of the algorithm can be performed in a constant number of time steps, usually in one time step, so that the τ-register only induces a constant slowdown compared to a standard `TAS` register. Nevertheless, there is a significant hardware overhead of $\mathcal{O}(\log n)$ additional registers and arithmetic logic units. It is therefore unlikely that such a register will be actually built, but it could be constructed based on this description.

D. Technical tools

In the technical parts of this paper we will be using the following version of the well-known Chernoff concentration inequality.

Lemma 1. *Let $X_1, \ldots, X_n$ be independent random variables such that $X_i \in \{0,1\}$. Let $p_i = P(X_i = 1) = \mathbb{E}[X_i]$, $X = \sum_{i=n}^{n} X_i$ and $\mu = \mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} p_i$. Then,*

1) *For any $\delta \in [0,1]$, $P[X \geq (1+\delta)\cdot\mu] \leq e^{-(\mu\delta^2)/3}$.*
2) *For any $\delta \geq 1$, $P[X \geq (1+\delta)\cdot\mu] \leq e^{-(\mu\delta)/3}$.*
3) *For any $\delta > 0$, $P[X \leq (1-\delta)\cdot\mu] \leq e^{-(\mu\delta^2)/3}$.*

III. Tight Renaming using $\log n$-Register

In this section we will design and analyse an algorithm for solving the tight renaming problem using $(\log n)$-registers in time $\mathcal{O}(\log n)$ and space $\mathcal{O}(n)$. We will now give a high-level description of the basic idea of algorithm and analysis.

We will be using an auxiliary array T_{aux} of length $2n$ of `TAS` bits belonging to $n/\log n$ many $(\log n)$-registers. Recall, each $(\log n)$-register has $2\log n$ `TAS` bits (which we also will refer to as *blocks*). We divide the array into $R = \mathcal{O}(\log n)$ clusters $C_1, C_2, \cdots, C_R$. C_i consists of c_i `TAS` bits with

$$c_i = n/(2c)^i$$

and c being a suitably large constant. Hence, the i-th cluster C_i contains

$$b_i = \frac{c_i}{2\log n} = \frac{n}{2\cdot(2c)^i\cdot\log n}$$

many $(\log n)$-registers and each of the registers is responsible for $\log n$ names.

The algorithm will proceed in rounds (for each given process), where for $i = 1, 2, \ldots$, in the i-th round all processes still *active* (initially for $i = 1$ all n of them) will randomly pick one `TAS` bit from *cluster* C_i. Each `TAS` bit having received at least one request will accept an arbitrary one of those. Each $(\log n)$-register will keep at most $\log n$ many successful requests (we refer to this as the *block discarding step*). A request which was successful will become *inactive*.

Each of the b_1 many $(\log n)$-registers in cluster C_1 will receive $n/b_1 = n/(n/2(2c)\log n) = 4c\log n$ many process requests in expectation, and at least $2c\log n$ of those with high probability. We will show that with high probability, in each $(\log n)$-register at least half the `TAS` bits will receive at least one process request, so that after the block discarding step we will have *precisely* $\log n$ accepted requests per block, *w.h.p.* In other words, the idea is to choose cluster sizes such that *w.h.p.* each block in a given cluster (round) receives just sufficiently many requests.

The remaining active processes will participate in the next round. In the final round, when we are down to a cluster size of $2\log n$ belonging to one $(\log n)$-register, the processes will access each of the `TAS` bits and eventually find a free `TAS` bit.

We should like to point out that whilst we talk about *rounds* as though we have a synchronised protocol this is in fact not the case. We use the notion only for ease of presentation. In reality, each process first tries cluster C_1, then cluster C_2, and so forth, until successful. In this sense the processes do operate in phases as indicated, but quite independent of one another.

Definition 2.

1) *Let*
$$R = \frac{\log(n) - \log\log(n) - 1}{\log(c) + 1}$$
(number of rounds).
2) *For $1 \leq i \leq R$, let*
$$c_i = n/(2c)^i$$
and
$$b_i = \frac{c_i}{2\log n}.$$
(cluster size and number of blocks in round i).
3) *For $1 \leq i \leq R$, let*
$$c'_i = \sum_{j=1}^{i-1} c_j$$
and
$$C_i = T_{aux}[c'_{i-1}+1, \ldots, c'_{i-1}+c_i]$$
(the i-th cluster in T_{aux}).
4) *For $1 \leq i \leq R$ and $1 \leq j \leq b_i$, let*
$$B_{i,j} = T_{aux}[c'_{i-1}+1+(j-1)\cdot b_i, \ldots, c'_{i-1}+1+j\cdot b_i-1]$$
(the j-th block in the i-th cluster in T_{aux}).

The main lemma of this section will use the following straightforward application of Chernoff's.

Lemma 3. *Let ℓ be an arbitrary, positive constant. Let $c \geq \max\{\ln 2, 2\ell+2\}$. Suppose $2c\log(n)$ balls are allocated i.u.r. into $2\log(n)$ bins. With probability at least $1 - 1/n^\ell$, there are no more than $\log(n)$ empty bins.*

Proof: For $1 \leq i \leq 2\log(n)$, let X_i be a binary random variable with $X_i = 1$ if and only if the i-th bin remains empty. Let

$$X = \sum_{i=1}^{2\log(n)} X_i$$

denote the number of empty bins and

$$\mu = \mathbb{E}[X].$$

For $1 \leq i \leq 2\log(n)$ we have

$$\mathbb{E}[X_i] = \mathbf{Pr}[X_i = 1] = \left(1 - \frac{1}{2\log(n)}\right)^{2c\log(n)} < 1/e^c$$

and therefore $\mu < 2\log(n)/e^c$. Choose δ such that

$$(1+\delta)\cdot\frac{2\log(n)}{e^c} = \log(n),$$

that is, $\delta = e^c/2 - 1$. Notice that $c > \ln(2)$ implies $\delta = e^c/2 - 1 > 0$. We wish to apply a Chernoff-type bound to X, but clearly the X_i are not independent. It is, however, well known (see e.g. Theorem 46 on page 21 of [14]) that they are *negatively associated*, which immediately implies that we may use any Chernoff bound (normally requiring independent random variables) of our choosing. Intuitively, negative association of a collection of random variables means that if we know some subset of the variables to have "large" values, then this decreases the probability of another, disjoint subset to take "large" values as well – in our case, if a subset of bins remains empty (with their $X_i = 1$) then another subset is less likely to remain empty as well (with *their* $X_i = 1$).

The remainder of the proof is now a mere formality. We use the generic version of Chernoff's, using our choice of δ from above, giving

$$\begin{aligned}
&\mathbf{Pr}\left[X \geq \log(n)\right] \\
&= \mathbf{Pr}\left[X \geq (1+\delta)2\log(n)/e^c\right] \\
&\leq \left(\frac{e^{(e^c/2-1)}}{(e^c/2)^{(e^c/2)}}\right)^{2\log(n)/e^c} \\
&= \left(\frac{e^{(e^c/2-1)2/e^c}}{(e^c/2)^{(e^c/2)2/e^c}}\right)^{\log(n)} = \left(\frac{e^{(1-2/e^c)}}{e^c/2}\right)^{\log(n)} \\
&= \left(\frac{2}{e^{c-1+2/e^c}}\right)^{\log(n)}.
\end{aligned}$$

Our constraint on c in the statement of this lemma implies

$$e^{c-1+2/e^c} > e^{c-1} > e^{(2\ell+2)-1} = e^{2\ell+1} > 2^{2\ell+1}$$

and therefore

$$\mathbf{Pr}\left[X \geq \log(n)\right] \leq \left(\frac{2}{2^{2\ell+1}}\right)^{\log(n)} = \left(2^{-2\ell}\right)^{\log(n)} < 1/n^{\ell}.$$

■

We are now ready to state and prove the main lemma of this section.

Lemma 4.

1) *In round R, we have a cluster size of $c_R = 2\log n$.*
2) *In each round, each block of size $2\log n$ receives $4c\log n$ requests in expectation, and at least $2c\log n$* w.h.p.

Proof:

1) As per Def. 2(2) we have $c_i = n/(2c)^i$. We solve $n/(2c)^i = 2\log n$ for i:

$$\begin{aligned}
&\frac{n}{(2c)^i} = 2\log n \;\Leftrightarrow\; \frac{n}{2\log n} = (2c)^i \\
&\Leftrightarrow \frac{\log(n) - \log\log n - 1}{\log(2c)} = i \\
&\Leftrightarrow \frac{\log(n) - \log\log(n) - 1}{\log(c)+1} = i.
\end{aligned}$$

This is exactly what we defined R to be in Def. 2(1).

2) By induction on the round.

In round 1 we have $c_1 = n/(2c)^1 = n/2c$ and $b_1 = n/(4c\log n)$. We throw n balls into cluster C_1, and therefore each of the b_1 many blocks receives $4c\log n$ requests in expectation. A simple application of Chernoff's shows that each block will receive $2c\log n$ requests *w.h.p.* According to Lemma 3, this implies that half of the $2\log n$ `TAS` bits in each block will receive at least one request. Consequently, after the block discarding step, precisely $\log(n)$ of the $2\log n$ `TAS` bits in each block will have accepted a request.

Suppose that up to and including round r, $1 \leq r < R$, each block of size $2\log n$ has had exactly $\log(n)$ many `TAS` bits accept a request each in their respective clusters, for a total of

$$\begin{aligned}
\sum_{i=1}^{r} b_i \log n &= \sum_{i=1}^{r} \frac{n\log n}{2\cdot(2c)^i\cdot\log n} \\
&= \sum_{i=1}^{r} \frac{n}{2\cdot(2c)^i} = \frac{n}{2}\cdot\frac{1-(1/2c)^r}{2c-1}
\end{aligned}$$

accepted and

$$\begin{aligned}
\rho_{r+1} &:= n - \frac{n}{2}\cdot\frac{1-(1/2c)^r}{2c-1} \\
&= \left(1 - \frac{1-(1/2c)^r}{2(2c-1)}\right)\cdot n
\end{aligned}$$

remaining active processes. In round $r+1$, those ρ_{r+1} processes will request bits in cluster C_{r+1} of size $c_{r+1} = n/(2c)^{r+1}$. This cluster contains $b_{r+1} = \frac{n}{2\cdot(2c)^{r+1}\cdot\log n}$ many blocks of size $\log n$ each. Each

such block will therefore receive

$$\begin{aligned}
\frac{\rho_{r+1}}{b_{r+1}} &= \left(1 - \frac{1-(1/2c)^r}{2(2c-1)}\right) \cdot \frac{n \cdot 2 \cdot (2c)^{r+1} \cdot \log n}{n} \\
&= \left(1 - \frac{1-(1/2c)^r}{2(2c-1)}\right) \cdot 2 \cdot (2c)^{r+1} \cdot \log n \\
&= \left(2 \cdot (2c)^{r+1} - \frac{1-(1/2c)^r}{2(2c-1)} \cdot 2 \cdot (2c)^{r+1}\right) \cdot \log n \\
&= \left(2 \cdot (2c)^{r+1} - \frac{1-(1/2c)^r}{2c-1} \cdot (2c)^{r+1}\right) \cdot \log n \\
&= (2c)^{r+1} \cdot \left(2 - \frac{1-(1/2c)^r}{2c-1}\right) \cdot \log n \\
&\geq (2c)^{r+1} \cdot \log n \\
&\geq 4c \log n
\end{aligned}$$

requests in expectation (notice that $\frac{1-(1/2c)^r}{2c-1} \leq 1$ and hence $2 - \frac{1-(1/2c)^r}{2c-1} \geq 1$). We may apply Chernoff's and find that at least $2c \log n$ requests arrive in each block of cluster C_{r+1}, *w.h.p.* Again, according to Lemma 3, this implies that half of the $2 \log n$ `TAS` bits in each block will receive at least one request.

A union bound over all rounds etc. proves the claim.

■

We can now state the main theorem of this section.

Theorem 5. *With high probability, the protocol as described in this section assigns n processes to a name space of size n in time $\mathcal{O}(\log n)$, using $\mathcal{O}(n)$ extra space.*

IV. Loose Renaming in the Standard Model

In this section we consider the problem of loose renaming where the name space is larger than n. In [8] the authors propose a $\mathcal{O}(\log \log n)$-time loose renaming algorithm that uses a name space of size $(1+\epsilon) \cdot n$ where $\epsilon > 0$ is an arbitrary constant. In this section we consider renaming algorithms using smaller name spaces and assume the model that was introduced in [8].

In this model the processes have access to standard asynchronous shared memory. They share registers which contain the names and on which they can perform `TAS` operations implemented in hardware. The algorithms are composed of steps. Any number of processes may fail by crashing, and a failed process does not perform further steps in the execution. The order in which the processes perform their accesses and the crashes are controlled by an adaptive adversary. The adversary is allowed to see the state of all processes (including the results of coin flips) when making its scheduling choices.

First we present some algorithms that rename most but not all of n processes using a name space of size n. To assign a name to all processes we apply the method of [8] and assign to the remaining processes names from a name space starting at $n+1$. We call a renaming algorithm *k-almost tight* if it assigns a name to all but $n-k$ processes, with $k = o(n)$.

Note that one can also apply the framework of [8] to transform our algorithms into adaptive algorithms when the number of active processes that are looking for a name is not known in advance. Unfortunately, the name space would become $\mathcal{O}((1+\epsilon) \cdot k)$, hence using our protocols would not result in an improvement compared to [8].

Lemma 6. *Assume we have n test-and-set registers and n processes. Then $\frac{n}{(\log \log n)^\ell}$-almost tight renaming can be done w.h.p. in the adaptive adversary model with a step complexity of*

$$\mathcal{O}((\log \log n)^\ell).$$

Proof: The algorithm works in $\ell \log \log \log n$ many rounds. Round i has 2^i many steps. In every step of each round, all unnamed processes send a request to a randomly chosen test-and-set register. Registers receiving a request are set by an arbitrary one of the accessing processes. The corresponding process has a name now and becomes *inactive*. Note that, if a register is set, it remains set for the rest of the algorithm. The total runtime of the algorithm (which is the number of steps) is

$$\sum_{i=1}^{\ell \log \log \log n} 2^i \leq (\log \log n)^\ell$$

We call round i *successful* if, at the end of round i, there are at most $n/2^i$ processes that are not assigned to a register. If all $\ell \cdot \log \log \log n$ rounds are successful there will be

$$\frac{n}{2^{\ell \cdot \log \log \log n}} = 2 \cdot \frac{n}{(\log \log n)^\ell}$$

processes left which are not assigned to a name.

In the following we prove by contradiction that every round is w.h.p. successful. Fix a round i, $1 \leq i \leq \ell \cdot \log \log \log n$, and assume round that i is the first round which is not successful. We can assume that during every step of round i we have at least $n/2^i$ active processes (otherwise round i is successful) and unset registers. Hence, the total number of random choices in round i is at least

$$\frac{n}{2^i} \cdot 2^i = n.$$

The probability that an arbitrary unset register does not receive any of the requests is at most

$$\left(1 - \frac{1}{n}\right)^n \leq \frac{1}{e}.$$

Let X_i be the random variable that counts the number of unset registers at the end of round i. Then

$$\mathbb{E}[X_i] \leq \frac{n}{2^{i-1}} \cdot \left(\frac{1}{e}\right).$$

Since $E[X_i] \geq n/\log\log\log n$, we can use Chernoff bounds to show that *w.h.p.* the number of unset registers at the end of the round is at most $\frac{n}{2^i}$, meaning that the round is successful. Now we can use the union bound over all rounds i to show the lemma. ∎

Corollary 7. *Assume we have $n + 2n/(\log\log n)^\ell$ test-and-set registers and n processes. Then,* w.h.p.*, loose renaming can be done in the adaptive adversary model with a step complexity of*

$$\mathcal{O}((\log\log n)^\ell).$$

Proof: First we use Lemma 6 to assign a name to all but $n/(\log\log n)^\ell$ many of the processes. Then we use the algorithm of [8] on the name space $n+1$ to $n + 2n/(\log\log n)^\ell$ to assign a name to the remaining unnamed processes. ∎

Lemma 8. *Assume we have n test-and-set registers and n processes. Then,* w.h.p.*, $n/(\log n)^\ell$-almost tight renaming can be done in the adaptive adversary model with a step complexity of*

$$2\ell \cdot (\log\log n)^2.$$

Proof: This algorithm works in $\log\log n$ phases. The registers are now divided into a sequence of clusters. For $1 \leq j \leq \log\log n$ the jth cluster contains $n/2^j$ many registers. In phase i the processes randomly choose registers from the ith cluster only. Every round consists of $2\ell \cdot \log\log n$ many steps. In every step of every round i, all unassigned processes send a request to a randomly chosen test-and-set register from cluster i. Registers receiving a request are set by one of the accessing processes, and the corresponding process becomes inactive.

At the beginning of round $i \geq 2$ there are at least

$$n - \sum_{j=1}^{i-1} \frac{n}{2^j} = \frac{n}{2^{i-1}}$$

many active processes. The probability for a node in cluster j to remain empty is

$$\left(1 - \frac{2^j}{n}\right)^{n/2^j \cdot 2\ell\log\log n} \leq \frac{1}{(\log n)^{2\ell}}.$$

Hence, the expected number of empty modes (which equals the expected number of not named processes) is $n/(\log n)^{2\ell}$. The result follows now from an application of Chernoff bounds. ∎

Corollary 9. *Assume, we have $n + 2 \cdot \frac{n}{(\log n)^\ell}$ test-and-set registers and n processes. Then,* w.h.p.*, loose renaming can be done in the adaptive adversary model with a step complexity of*

$$\mathcal{O}((\log\log n)^2).$$

Proof: Similar to the proof of Corollary 7. ∎

V. Conclusion

In this paper we have considered the renaming problem in the asynchronous shared memory model. By utilizing new hardware features and extending the concept of the test-and-set register, we have shown that even a fairly straightforward randomized algorithm can perform tight renaming in $\mathcal{O}(\log n)$ steps with high probability. The hardware added is a set of register clusters, each containing $\log n$ names, which increase the success probability for the random accesses of the processes by seemingly enlarging the name space.

Our solutions to the loose renaming problem work in the standard model in which the names are stored in "plain" test-and-set registers. The algorithms are the first to achieve almost tight renaming in poly-double-logarithmic time mapping n names to a namespace of size only $(1 + o(1)) \cdot n$.

While there is a known matching lower bound for loose renaming, it remains open to show that the lower bound for tight renaming can be extended to the τ-register. An interesting future task will be the exploration of modern hardware capabilities and how new features can improve solutions to the fundamental problems in distributed computing.

References

[1] S. Chaudhuri, M. Herlihy, and M. R. Tuttle, "Wait-free implementations in message-passing systems." *Theor. Comput. Sci.*, vol. 220, no. 1, pp. 211–245, 1999.

[2] M. Okun, "Strong order-preserving renaming in the synchronous message passing model." *Theor. Comput. Sci.*, vol. 411, no. 40-42, pp. 3787–3794, 2010.

[3] D. Alistarh, O. Denysyuk, L. Rodrigues, and N. Shavit, "Balls-into-leaves: Sub-logarithmic renaming in synchronous message-passing systems," in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC '14. New York, NY, USA: ACM, 2014, pp. 232–241.

[4] D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu, and R. Guerraoui, "Fast randomized test-and-set and renaming." in *DISC*, ser. Lecture Notes in Computer Science, N. A. Lynch and A. A. Shvartsman, Eds., vol. 6343. Springer, 2010, pp. 94–108.

[5] A. Castañeda, S. Rajsbaum, and M. Raynal, "The renaming problem in shared memory systems: An introduction," *Comput. Sci. Rev.*, vol. 5, no. 3, pp. 229–251, Aug. 2011.

[6] D. Alistarh, J. Aspnes, S. Gilbert, and R. Guerraoui, "The complexity of renaming," in *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, ser. FOCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 718–727.

[7] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, and M. Zadimoghaddam, "Optimal-time adaptive strong renaming, with applications to counting." in *PODC*, C. Gavoille and P. Fraigniaud, Eds. ACM, 2011, pp. 239–248.

[8] D. Alistarh, J. Aspnes, G. Giakkoupis, and P. Woelfel, "Randomized loose renaming in O(log log n) time," in *Proceedings of the 2013 ACM Symposium on Principles of distributed computing*, ser. PODC '13. New York, NY, USA: ACM, 2013, pp. 200–209.

[9] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, and R. Guerraoui, "Tight bounds for asynchronous renaming," *J. ACM*, vol. 61, no. 3, pp. 1–51, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2597630

[10] A. Brdosky, F. Ellen, and P. Woelfel, "Fully-adaptive algorithms for long-lived renaming," *Distributed Computing*, vol. 24, no. 2, pp. 119–134, 2011.

[11] A. Panconesi, M. Papatriantafilou, P. Tsigas, and P. M. B. Vitanyi, "Randomized naming using wait-free shared variables," *Distributed Computing*, vol. 11, no. 3, 1998.

[12] G. Giakkoupis and P. Woelfel, "On the time and space complexity of randomized test-and-set," in *PODC*, D. Kowalski and A. Panconesi, Eds. ACM, 2012, pp. 19–28.

[13] W. Eberly, L. Higham, and J. Warpechowska-Gruca, "Long-lived, fast, waitfree renaming with optimal name space and high throughput," in *DISC*, ser. Lecture Notes in Computer Science, S. Kutten, Ed., vol. 1499. Springer, 1998, pp. 149–160.

[14] D. Dubhashi and D. Ranjan, *Balls and Bins: A Study in Negative Dependence*. BRICS, 1996. [Online]. Available: http://www.brics.dk/RS/96/25/BRICS-RS-96-25.pdf